hooks

19 pag.

¡Hooks al Ataque!

Week	Week 1
Unit/Module	Unit 1
Files	useContext.pdf useEffect.pdf useReducer - Paso a paso.pdf useRef - Casos de uso.pdf

Hooks

Estos surgieron como respuesta a ciertas problemáticas que presentaba el uso de componentes de clase en React, como la dificultad para comprender componentes complejos y manejar y reutilizar la lógica de estado.

Implementación de Hooks

Los Hooks son una funcionalidad de React que nos permite utilizar el estado y otras características sin la necesidad de escribir componentes de clase.

> 💡 **Gracias a los Hooks, podemos extraer lógica de estado de un componente de forma tal que pueda ser probado y reusado independientemente. Los Hooks nos permiten reutilizar lógica de estado sin cambiar la jerarquía del componente, lo que facilita el compartir Hooks entre muchos componentes o incluso con la comunidad.**

Además, los Hooks nos permiten dividir un componente en funciones más pequeñas basadas en las piezas relacionadas, en lugar de forzar una división basada en los

métodos del ciclo de vida. Esto nos permite manejar estados locales dentro de cada componente, lo que los hace más predecibles.

Hooks nativos en React

- **useState**: Declara variables de estado y las actualiza.
- **useEffect**: Permite incluir y manejar efectos secundarios.
- **useContext**: Permite crear un proveedor de datos globales para que estos puedan ser consumidos por los componentes envueltos por el proveedor
- **useReducer**: Es una variación de useState, resulta conveniente utilizarlo cuando nos encontramos ante un gran número de piezas de estado.
- **useImperativeHandle**: Personaliza el valor de instancia que se expone a los componentes padres cuando se usa ref.
- **useMemo**: Devuelve un valor memorizado. También es útil para evitar renderizados innecesarios. Optimiza el rendimiento.
- **useRef**: Permite utilizar programación imperativa. Devuelve un objeto mutable cuyo cambios de valor no volverán a renderizar el componente. Por el contrario, se mantendrá persistente durante la vida completa del componente.
- **useCallback**: Devuelve un callback que es memorizado para evitar renderizados innecesarios.
- **useLayoutEffect**: Es similar a useEffect, pero se dispara de forma síncrona después de todas las mutaciones de DOM.
- **useDebugValue**: Permite aplicar una etiqueta a los hooks personalizados visible mediante React DevTools.

---

useState

En React, la interfaz de usuario representa el estado en el momento de renderización inicial. Si alguna pieza de ese estado actualiza su valor, React volverá a renderizar los componentes involucrados con dicha pieza de estado tras hacer una comparación entre el virtual DOM y el DOM guardado en memoria. El objetivo de esto es que la actualización sea eficiente y basada en la diferencia entre ambos DOM.

Desde el lanzamiento de Hooks, los componentes funcionales pueden declarar y actualizar su estado interno mediante el Hook

> 💡 ***useState*** *es una función que crea variables de estado y las actualiza.*

¿Cómo funciona?

```
const [job, setJob] = useState("Seeking");
```

Cada componente tiene asociado una lista interna de “celdas de memoria”. Estas celdas de memoria se traducen en objetos literales de JavaScript en donde incluimos nuestros datos. Cuando invocamos useState, la función va leyendo las celdas una a una. De este modo, múltiples llamados a useState obtienen múltiples estados locales independientes.

En teoría, esto nos permitiría declarar infinitas const de useState.

```
const [count, setCount] = useState(0);
const [phone, setPhone] = useState("");
const [username, setUsername] = useState("");
const [email, setEmail] = useState("");
const [password, setPassword] = useState("");
const [confirmPassword, setConfirmPassword] = useState("");
const [otaku, setOtaku] = useState(false);
const [loveLevel, setLoveLevel] = useState({day: "orange", night: "blue"});
```

Podemos incluir useState las veces que necesitemos, la única condición es que sea llamada desde un nivel superior de código, no en un bloque.

❌Ejemplo incorrecto:

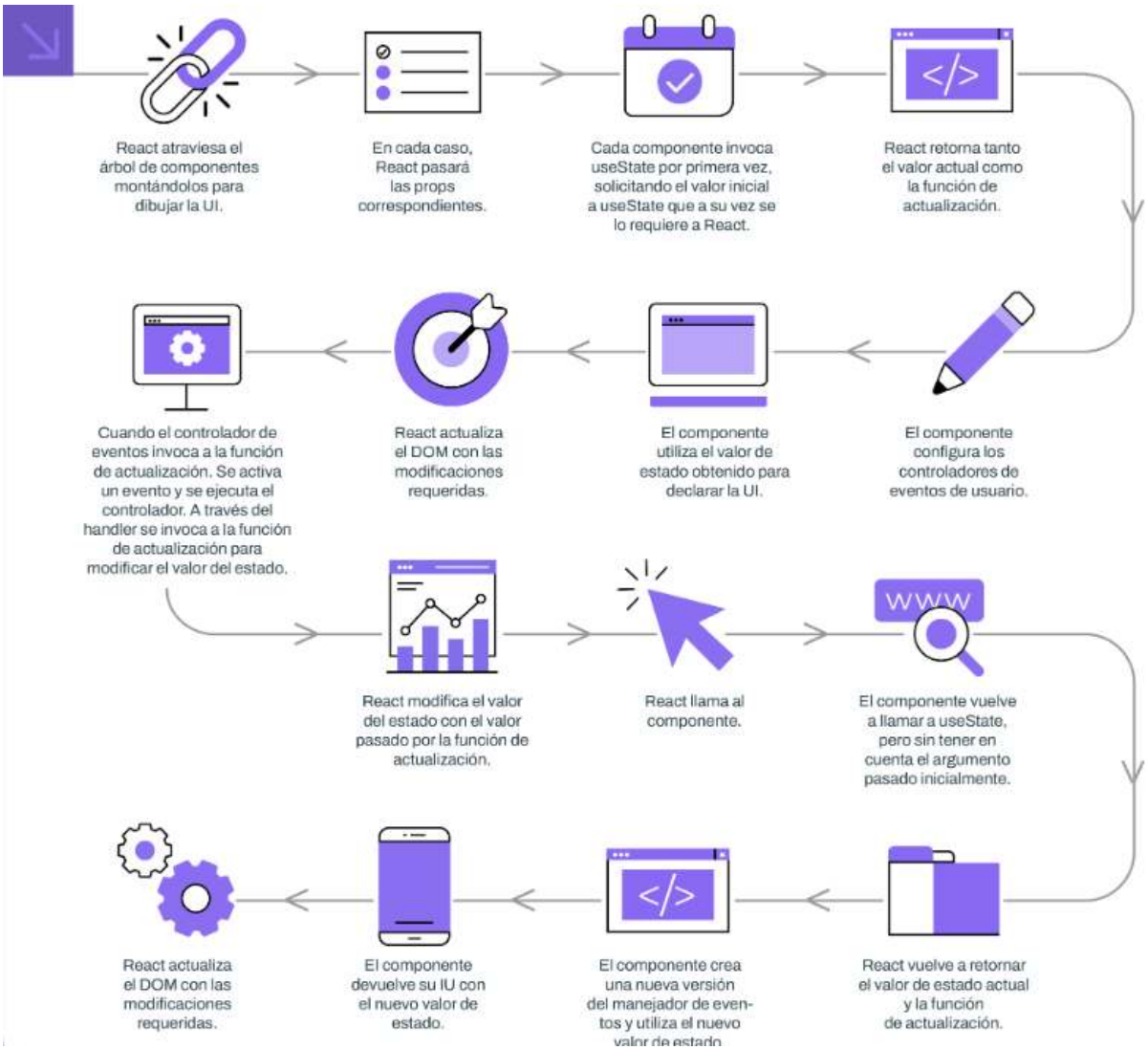
```
Const Counter = () => {
    const estoEstaMal = () => {
        const [count, setCount] = useState(0);                    (incorrecto)
```

```
    }
    return {estoEstaMal};
  };
```

✅Ejemplo correcto:

```
const Counter = () => {
   const [count, setCount] = useState(0);                    (Correcto)
   return {count};
};
```

Ciclo useState



useEffect

> 💡 ***useEffect*** *es una función que permite incluir y manejar efectos secundarios.*

Es el Hook que se encarga de configurar y gestionar los efectos secundarios en el ciclo de vida de un componente funcional. Los hace:

- Predecibles.
- Controlables

Es el equivalente - sintetizado en una sola función - de lo que podemos hacer con los métodos componentDidMount, componentDidUpdate y componentWillUnmount en un componente de clase.

Características:

- Tiene acceso a las variables dentro del componente, puesto que comparten el mismo ámbito.
- Se ejecuta después del primer render y luego de cada render posterior (si no lo configuramos en sentido contrario)
- Nos permite pensar de otro modo al ciclo de vida, en términos de sincronización de piezas lógicas.

Casos de uso

- Establecer el título de una página de forma imperativa:

```
import {useEffect, useState} from "react";

const ChangeTitleExample = ()  ⇒ {
  const [title, setTitle] = useState (
  "Certified Tech Developer | Digital House"
);
```

```
  useEffect(() => {
    document.title = title;
}, [title]);

  return <div />;
};
```

- Trabajar con temporizadores: setInterval o setTimeout:

```
import { useEffect, useState} from "react";

const Interval = () => {
  const [cantidad, setCantidad] = useState(2);

  useEffect(() => {
    const interval = setInterval(() => {
      setCantidad((prevState) => ++prevState);
  }, 1000);
  }, []);

  return <div>Quiero {cantidad} de chocolates </div>
  };

  export default Interval;
```

- Leer ancho, alto o posición de elementos del DOM.
- Registrar mensajes (console.log)

```
import { useEffect } from "react";

const GetResizeFromWindow = () => {
  useEffect(() => {
    function handleResize() {
      console.log(
        "Redimensionar: ",
        window.innerWidth,
        "x",
        window.innerHeight
    );
  }
  window.addEventListener("resize", handleResize);
});

return <div />
};

export default GetResizeFromWindow;
```

- Establecer u obtener valores de almacenamiento local

```
import React, {useState, useEffect } from "react";

import Login from "./Login";
import Home from "./Home";

const UserLogged = () => {
  const [user, setUser] = useState();

useEffect(() => {
  function checkUser() {
    const item = localStorage.getItem("User");
    if (item) setUser(item);
  }

  window.addEventListener("storage", checkUser);

  return () => {
    window.removeEventListener("storage", checkUser);
  };
}, []);

  return <>{user ? <Home /> : <Login />} </>;
};

export default UserLogged;
```

- Realizar peticiones a servicios.

```
useEffect(() => {
  async function getCandy() {
    const resp = await fetch("https://alotofcandy.iumi");
    const data = await resp.json();
    setCandy(data);
}
getCandy();
}, []);
```

useContext

> 💡 ***useContext** es una función que permite crear un proveedor de datos globales para que estos puedan ser consumidos por los componentes envueltos por el proveedor.*

Context

Context nos permite encapsular datos e inyectarlos en componentes incluidos dentro de dicho contexto.

Contenedor de componentes:

- Todo valor definido y actualizado en el contexto podrá ser usado por los componentes incluidos dentro de tal contexto.
- En este sentido, si bien Context brinda a la aplicación de cierta globalidad en los datos, es una globalidad parcial.
- Esto no es malo, por el contrario, es una característica superútil ya que nos permite definir el alcance específico de tales globalidades.

> 💡 Context nos evita la maldición de pasar datos a través de los árboles de componentes enteros mediante props.

Ej: Ambos subarboles de componentes (Left y Main) podrán consumir el contexto TitleContext.

```
import React from "react";

class App extends React.Component {
  render() {
    return (
      <TitleContext.Provider value="Soy un título">
        <Left />
        <Main />
      </TitleContext.Provider>
      );
```

```
  }
}
```

Casos de uso

1. **Manejo de theming**: Implementación de modo oscuro o temas alternativos.
2. **Autenticación de usuarios**: Autenticar los usuarios en el servicio.
3. **Internacionalización**: La selección de idioma preferido.

Pasos para la implementación de Context

1. Creación mediante el método createContext.
2. Usar un Context Provider para encapsular un árbol de componentes.
3. Poner cualquier valor en el Provider usando la propiedad value.
4. Leer esta propiedad en cualquier componente encapsulado usando el Context Consumer.

createContext

Un objeto Context se crea pasando por defecto un valor al método de React createContext. Si bien createContext acepta un valor inicial, este no es obligatorio.

Como resultado de crear el contexto, obtendremos dos propiedades que, a la vez, se declaran como componentes: **Provider** y **Consumer**.

Ejemplo:

```
import {createContext} from "react";

const peopleMood = "Happy";

export const PeopleContext = createContext(peopleMood);
```

Provider

El contexto debe ser global para los componentes que lo consumen. Es por esto que con Context nos brinda un Provider. Este es un componente que toma una propiedad llamada **value**, es decir, el valor dado al contexto y lo inyecta a los componentes englobados por él. Este valor puede ser el que se desee, hasta un objeto con múltiples valores.

No todos los componentes descendientes del contexto estarán suscritos al mismo, salvo que declaren la suscripción a este. Esto es debido a que no todos los componentes necesitarán estos datos.

Cuando un componente suscrito se renderice, leerá el valor actual del contexto provisto por el Provider. Por lo que la responsabilidad del Provider será dar acceso a los datos almacenados en el contexto.

```
<PeopleContext.Provider value="mucha>
  <World />
</PeopleContext.Provider>
```

Consumer

Para que un componente pueda recibir el valor de un contexto, el componente debe estar suscrito al mismo. La suscripción se realiza mediante el componente **Consumer**.

- Este componente deberá incluir una función como children.
- Esta función recibirá el argumento **value** con el valor actual del contexto, retornará un nodo React y estará a la escucha de cambios.
- Respecto al árbol de componentes, el Consumer estará debajo del Provider. Esta función será llamada en el momento de renderizar.
- Debemos pensar a Context como un canal para compartir datos.
- Por cada Provider habrá un Consumer que solo se relacionarán entre sí.

```
  import React, {Component} from "react";
import {ConsumerDeLaVida} from "./ContextDeLaVida";

class App extends Component {
  render()
    return (
      <ConsumerDeLaVida>
        {(props) => {
          return <div>{props.laVida}</div>
        }}
      </ConsumerDeLaVida>
      );
  }
}
```

---

useRef

> 💡 **useRef es un Hook adicional que crea (y devuelve) una referencia que podemos asignar a un elemento del DOM para... ¡referenciarlo!** 😅

Cuando creamos una **referencia**, esta poseerá una propiedad llamada **current**, que contiene el contenido de la actual de **referencia.**

¿Cómo lo usamos?

Lo único que debemos hacer es crear la referencia (ref):

```
const ref = useRef(null);
```

Y luego anexamos esa referencia al elemento que deseemos:

```
<button ref="{ref}">Soy un botón</button>
```

Ahora **ref** contiene una propiedad especial: **ref.current**, un valor mutable DOM del elemento que anexamos.

💡 **¿Por qué se agrega un null como parámetro de useRef? Es porque useRef toma un valor inicial que luego asignará a ref.current.**

✅**Es buena práctica indicar el null porque al momento de declarar el Hook no estamos asociando la referencia a ningún elemento del DOM. Entonces, si un bloque de código depende de nuestro useRef, podemos garantizar que será null o poseerá ref.current.**

Casos de uso

- **Crear una referencia a un elemento**:
  - **onChange:** para ejecutar una función cuando cambia el valor de un input.
  - **onBlur:** si queremos detectar si el usuario hizo foco en nuestro elemento.
  - **onClick:** si queremos capturar cuando un usuario hace clic en nuestro elemento… y otros ejemplos más.

  ❌ En JavaScript puro podemos acceder a estos elementos con **document.getElementById** o **document.querySelector** para luego realizar las funciones correspondientes.

  El inconveniente con React es que, al manejar un DOM virtual y estar trabajando con componentes, la cosa se vuelve un poco más engorrosa. ¿Por qué? Porque perdemos la garantía de que nuestro ID no sea repetido, entonces, **getElementById** puede fallar al buscar el elemento que deseamos. También pueden aparecer bugs inesperados.

  ✅**useRef** viene a salvarnos el día creando una referencia única en nuestro código, independiente de ID, class o cualquier identificador que utilicemos.

    - Usamos ID para identificar elementos en nuestra página web con JavaScript puro.
    - React no garantiza que nuestros IDs sean únicos: buscarlos por ID es casi imposible.

- **useRef** crea una referencia (un ID de React) y lo asigna a un elemento, para luego ser utilizado.

- **Mantener el valor entre renderizados**: **useRef** nos permitirá modificar dicho estado a través de la propiedad **current** sin la necesidad de producir un nuevo render.

useReducer

> 💡 **useReducer es una alternativa a useState. Maneja estados de otra forma: nos ofrece una estructura más robusta para manejar estados más complejos. Por ejemplo, un objeto con subvalores que deben actualizarse de manera independiente.**

⚠️ **Importante**:

- useReducer no es de ninguna forma un reemplazo a useState, ni tampoco debe considerarse la primera opción al manejar el estado en un componente. Por lo general, se lo considera más adelante, cuando el estado crece. Crear una prueba de concepto usando useReducer ayuda mucho a entender si mejora o empeora la legibilidad.

¿Cómo lo usamos?

```
const [state, dispatch] = useReducer(reducer, initialState);
```

- state: Estado actual de nuestro useReducer completo (similar al primer elemento de **useState)**
- dispatch: Función que toma un objeto (similar al segundo elemento de **useState)**
- reducer: Función reductora, encargada de manipular el estado.

- initialState: Estado inicial (luego puede ser accedido en state). Es similar al primer parámetro de **useState.**

```
const reducer = (state, action) => {
  switch (action.type) {
    case "CAMBIAR_EDAD":
      // Aqui iria nuestra validacion de la edad
      return { ...state, edad: action.payload }
    default:
      return state;
  }
}
```

La estructura del objeto **action** es de uso frecuente. Podemos declarar las **action** como queramos. Por ejemplo, en lugar de llamarse **type**, que se llame **name**, y en lugar de **payload**, que se llame **data**. Sin embargo, hacerlo de esta manera sería ir en contra de los estándares que se generaron alrededor de estas herramientas (o puede que tu nuevo equipo use **reducer** de una manera distinta).

Lo importante es tener en claro que **action** es un objeto, que debe poseer un identificador (qué queremos actualizar) y un valor (con qué valor lo actualizamos).

La función reducer se ejecutará cada vez que llamemos a la función **dispatch**, actualizando el estado (si es necesario). Siguiendo el ejemplo, lo llamaríamos de la siguiente forma:

```
dispatch({ type: "CAMBIAR_EDAD", payload: 25 });
```

Paso a paso del useReducer

Ejemplo de contador

```
import { useState, Fragment } from "react";
export default (props) => {
  let [conteo, setConteo] = useState(0);
  let handleIncrementClick = () => {
    setConteo((prevState) => prevState + 1);
  };
```

```
  let handleDecrementClick = () => {
    setConteo((prevState) => prevState - 1);
  };

  let handleResetClick = () => {
    setConteo(0);
};

  return (
    <>
      <h1>El conteo es {conteo}</h1>
      <div>
        <button
          onClick={() => {
            handleDecrementClick();
          }}
        >
          Decrementar
        </button>
        <button
          onClick={() => {
            handleIncrementClick();
          }}
        >
          Incrementar
        </button>
        <button onClick={() => handleResetClick()}>Resetear</button>
      </div>
    </>
  );
};
```

Tenemos tres métodos que manejan el estado conteo:

- Uno que incrementa.
- Otro que decrementa.
- Un último que resetea a 0 el número guardado.

Comencemos por reemplazar **useReducer** por **useState** en el **import**:

```
import { useReducer, Fragment } from "react";
export default (props) => {
  return (
    <>
      <h1>El conteo es ...</h1>
      <div>
        <button>Decrementar</button>
```

```
        <button>Incrementar</button>
        <button>Resetear</button>
      </div>
    </>
  );
};
```

1. **InitialState**:

Lo primero que vamos a agregar es una variable que contenga un objeto literal con el valor inicial del estado que vayamos a tener. Esta estará **por fuera del componente**.

Por ahora, y para facilitarnos la tarea, vamos a escribir en español: **estadoInicial** será nuestra variable.

```
import { useReducer, Fragment } from "react";
let estadoInicial = { conteo: 0 };
```

2. **Reducer, state y action**

Otra cosa que vamos a necesitar es una función que tenga la lógica para saber de qué manera modificar el estado: si tenemos que incrementar, decrementar o resetear. A esta función podemos llamarla **reducer** y la usaremos como callback al usar **useReducer**. **Esta función también irá por fuera del componente**.

Continuando con su anatomía, esta función reductora utilizará dos parámetros:

- El estado actual, al cual podemos llamar **estado**.
- Un objeto literal que tendrá información acerca de la acción que queremos hacer para el o los estados. A este parámetro lo llamaremos **accion**.

Finalmente, esta función reductora retornará un nuevo valor (probablemente un objeto con muchas propiedades) y este será el que quede actualizado como estado. Veamos cómo queda conformada esta función:

```
let estadoInicial = { conteo: 0 };
let funcionReductora = (estado, accion) => {
  // Vamos a poner un console.log para saber bien de qué se trata "estado" y "accion"
```

```
  console.log(estado, accion);
  return { conteo: 99 };
};
```

3. **Desestructuración de useReducer y Dispatch**

Ahora lo que nos queda es utilizar **useReducer** dentro del componente y desestructurar (*destructuring*) dos de los elementos que nos provee: el **estado** y la información para **“despachar”** o entregar. Al mismo tiempo, habiendo creado una función reductora y un estado inicial, esto es lo que vamos a pasar como parámetros al Hook.

Avancemos en el código para que quede claro:

```
let estadoInicial = { conteo: 0 };

let funcionReductora = (estado, accion) => {
  // Vamos a poner un console.log para saber bien de qué se trata "estado" y "accion"
  console.log(estado, accion);
  return { conteo: 99 };
};

function Contador(props) {
  const [estado, entregaDeInfo] = useReducer(funcionReductora, estadoInicial);
return (
    <>
      <h1>El conteo es ...</h1>
      <div>
        <button>Decrementar</button>
        <button>Incrementar</button>
        <button>Resetear</button>
      </div>
    </>
  );
}
```

4. **Sumar eventos y poner en funcionamiento:**

Vamos a hacer que el botón “Decrementar” vaya restando al estado **conteo**. Para eso, vamos a agregar el evento **onClick** y dentro ejecutaremos una función para “entregar información” a la función reductora. Ahora, si estás pensando que esta situación de “entregar información” es algo que ya de alguna manera preparamos previamente…

¡estás en lo correcto! **entregaDeInfo** es la función que utilizaremos para hacerlo: entregar o despachar información a través de un objeto literal.

```
<button
  onClick={() => {
    entregaDeInfo({ tipo: "decrementar" });
  }}
>
  Decrementar
</button>;
```

Utilizamos la función **entregaDeInfo** (previamente creada cuando desestructuramos en el paso 3) y le estamos pasando un objeto literal con una propiedad **tipo**. Aquí en general aclaramos cuál es el tipo de “acción” que queremos hacer para el o los estados de nuestro componente. Ahora, ¿en qué momento dijimos qué hacer cuando el tipo de acción sea “decrementar”? Vamos a volver a nuestra función reductora y darle un hermoso cierre a todo esto.

Dentro de nuestro reducer, teníamos un console.log y un valor de retorno. Por ahora, si hacemos clic en el botón decrementar y chequeamos el estado, nuestro valor pasará de ser 0 (valor inicial) a 99. Claro, nuestra función no sabe hacer otra cosa que retornar **{ conteo: 99 }**.

El toque mágico que agregaremos es que dentro de esta función vamos a poder saber el tipo de acción que estamos activando. ¿Estamos decrementando, incrementando o reseteando? Esto lo aclaramos en el tipo de entrega y podemos acceder a este dato dentro de nuestra función reductora en la propiedad **tipo** del parámetro **accion**. Solo nos falta evaluar:

- Si el tipo de acción es decrementar, restamos uno.
- Si el tipo de acción es incrementar, sumamos uno.
- Si el tipo de acción es resetear, igualamos a cero.

Veamos el código de nuestra función reductora:

```
let funcionReductora = (estado, accion) => {
  switch (accion.tipo) {
    case "incrementar":
```

```
      return { conteo: estado.conteo + 1 };
    case "decrementar":
      return { conteo: estado.conteo - 1 };
    case "resetear":
      return { conteo: 0 };
    default:
      throw new Error("No se ha recibido ningún tipo de acción...");
  }
};
```

Muchos componentes manejan estados algo más complejos: propiedades que dentro tienen objetos y propiedades anidadas. En ese sentido, **useReducer facilita el orden de estos elementos**.